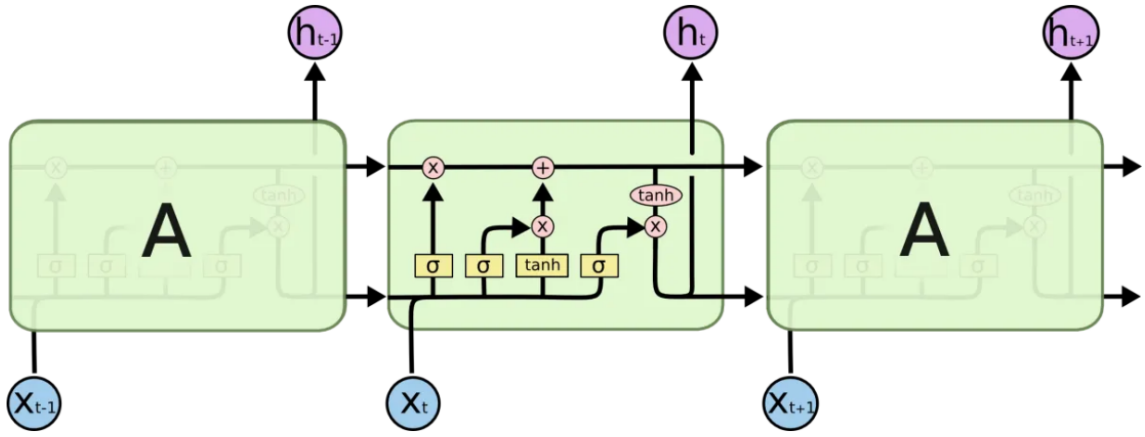
Hw2 report

Method in details

Build model

在這次的作業裡，我使用了 pytorch 中的 Long Short-Term Memory (LSTM)，架構大致如下：



我使用了以下參數來建構我的 LSTM 模型：

- input_size: 輸入 x 的特徵數量 (= 300)
- hidden_size: 隱藏層 h 的特徵數量 (= 256)
- num_layers: LSTM 的循環層數 (= 5)
- dropout: 每一層 LSTM 後都會 dropout (= 0.2)

以下是我的模型的程式碼：

```
class LSTM(torch.nn.Module):
    def __init__(self):
        super(LSTM, self).__init__()
        self.embed = torch.nn.Embedding.from_pretrained(torch.from_numpy(pretrained_weight).float())
        self.lstm = torch.nn.LSTM(input_size=dimension_model, hidden_size=hidden_size,
                                  num_layers=num_layers,dropout=dropout)
        self.linear = torch.nn.Linear(hidden_size, linear_hidden_size)
        self.fc = torch.nn.Linear(linear_hidden_size, classes)

    def attention(self, lstm_output, final_state):
        lstm_output = lstm_output.permute(1, 0, 2)
        merged_state = torch.mean(final_state, dim=0)
        merged_state = merged_state.unsqueeze(2)
        weights = torch.bmm(lstm_output, merged_state)
        weights = torch.nn.functional.softmax(weights.squeeze(2), dim=1).unsqueeze(2)
        ret = torch.bmm(torch.transpose(lstm_output, 1, 2), weights).squeeze(2)
        return ret

    def forward(self,data):
        x = self.embed(data)
        x,(h_n, c_n) = self.lstm(x.transpose(0, 1))
        x = self.attention(x, h_n)
        x = self.linear(x)
        x = self.fc(x)
        return x
```

另外，在我的模型中，我另外加入了 pretrain word embedding 和 attention 的機制，也就是說，在 embedding layer 是使用我預先訓練好的 weights，接著套用 pytorch 中的 LSTM，並引入 attention 的機制，使得當前的輸入與目標狀態越相似時，當前的輸入權重就會越大，最後透過線性的方式得到我要的答案。

關於 pretrain word embedding 和 attention 的機制，下方會有更詳細的說明。

Preprocess Method

原先在句子 preprocessing 的過程中，會依序執行下面五個步驟：

1. lower → 利用 `.lower()` 將所有字母轉成小寫
2. remove stopwords → 利用 `from nltk.corpus import stopwords` 來移除 stopwords
3. remove puntuation → 利用 `from string import punctuation` 來移除標點符號
4. lemmatization → 利用 `from nltk.stem import WordNetLemmatizer` 將動詞、名詞、形容詞都進行字型還原
5. remove digits → 利用 `from string import digits` 來移除數字

以此句子為例 → I went to the elementary school when I was 6 years old.

after lower → i went to the elementary school when i was 6 years old.

after removing stopwords → went elementary school 6 years old.

after removing puntuation → went elementary school 6 years old

after lemmaatization → go elementary school 6 year old

after removing digits → go elementary school year old

原先以為做完這些處理之後，準確率會提升，但似乎和我想得有些許的不同，在 kaggle 上的結果只有 0.19313，於是我做了個實驗想確認究竟是哪個 preprocess 的方法不適合用於此。

以下是我用 train_HW2dataset.csv 作為訓練資料，dev_HW2dataset.csv 作為驗證準確率的資料，所得到的準確率：

	accuracy
without preprocessing	58 %
lower	61 %
remove stopwords	51%
remove puntuation	46%
lemmatization	53%
remove digits	56 %

從上述表格可以發現，除了 lower 之外，其他無論是何種 preprocessing 的方法，都會使準確率降低。

其中，移除數字是影響最小的，我認為這是因為有數字的句子本身就不多，再加上數字通常是不代表任何情緒的，所以在這次的作業裡影響並不大。

至於移除標點符號對於準確率的影響，是四者裡面最大的，觀察資料集後，我發現有許多情緒其實是根據標點符號來表達的，像是這次有個情緒為 “surprise”，有許多句子就是根據後面的驚嘆號來決定此句子需標註為 ”surprise”。

另外，我認為移除 stopwords 會使準確率下降的原因是因為有些關鍵字(ex. not) 其實是需要存在的，如果將其移除，將會破壞整體的語意結構，進而影響到 model 在語意上的判斷。

一開始在處理句子 preprocessing 時，我原本想用 stemming 的方式，但後來發現有些字詞後的 ed 會直接被刪去，即便此字詞不為過去式，像是 indeed 經由 stemming 處理後會變成 inde，影響句子判斷，所以後來我便選擇 lemmatization。原先我認為此方式能解決一個原型動詞和其過去式會被視為兩個不同單詞的問題，沒想到準確率不但沒有提升，反倒還降低了不少。

Improve the accuracy on those emotion which are in small scale

在 train_HW2dataset.csv 加上 dev_HW2dataset.csv 裡，每個 label 所出現的個數和佔比如下表：

	個數	佔比
neutral	6560	0.4587
surprise	1700	0.1189
fear	392	0.0274
sadness	1021	0.0714
joy	2526	0.1766
disgust	395	0.0276
anger	1708	0.1194
total	14302	1.0

根據上表可以發現有接近一半的資料都被標記為 neutral，被標記為 fear 和 disgust 的分別都只有 2.7%。

一開始我希望能增加資料，於是將 dev_HW2dataset.csv 的資料加進來一起訓練，然而，為了解決資料不平衡的問題，避免每次都只挑到 neutral 這種類別的資料，我使用了 K-fold cross validation 來確保每筆資料都會被挑選到，並且都只會被挑選到一次，不會因為多次被挑選，而導致 overfitting。

以下是尚未使用 K-fold cross validation 和使用 K-fold cross validation 在 kaggle 上的分數：

尚未使用 K-fold cross validation

使用 K-fold cross validation

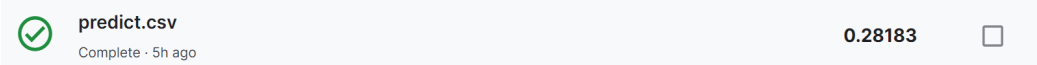
Pretrain word embedding

為了提升準確率，我使用了 pretrained word embedding 的技術，將每個單字都賦予一個向量，用來表示此單字在各種維度所表示的相對程度，從多面向來觀察。

在此次作業裡，我使用的是由 stanford 團隊所創造的 GloVe，一開始我使用的是 glove.6B.50d.txt，原先認為加上 pretrained word embedding，準確率必會提升，沒想到並不是如此，後來我想說可能是維度不夠，於是使用了 glove.6B.300d.txt，準確率確實提升了一些，但結果還是不盡理想，後來我才發現在 word2index 裡頭共有 6164 個單字，但卻只有 5348 個有在 glove.6B.300d.txt 裡頭，大約有 1/5 的單字是不在裡頭的。

為了讓剩餘 1/5 的單字也能被賦予詞向量，我使用了 glove.840B.300d.txt，6164 個單詞中，共有 5493 個有被賦予詞向量，和先前在 glove.6B.300d.txt 的結果相比，只增加了 145 個，但在 kaggle 上的分數也確實提升了一些

以下是使用 glove.6B.300d.txt 和 glove.840B.300d.txt 在 kaggle 上的分數：



pretrained word embedding with glove.6B.300d.txt on kaggle

pretrained word embedding with glove.840B.300d.txt on kaggle

在這部分，我只使用了 GloVe 來訓練我的模型，或許之後可以試著使用 Word2vec，裡面可能會有更多我所需要的單詞。

Attention on my model

如果缺乏 attntion 機制可能會有問題，因為整個輸入被 encode 成一個 vector，但不同部分其實有不同程度的重要性，例如 how is your weekend 裡，weekend 肯定比 is 對於理解來得重要。

attention 的機制是一個相似性的度量，當前的輸入如果與目標狀態越相似，那麼在當前輸入的權重就會越大，這說明了當前的輸出會更加依賴於當前的輸入，所以此機制就是在幫助 RNN 的輸出找到值得注意的地方，產生更相關的結果。

在我的模型當中，首先，我先引入計算機制，計算兩者的相似性和相關性，這部分我使用的方式是求兩者的向量內積，得到當前最關注的部分。接著，計算 softmax，將前一步所計算的值轉成 probability distribution，計算輸入中 hidden 的平均權重，最後拿 hidden state 和 vector 一起進行預測，如此一來，便可以注意到輸入跟自己最相關的部分，做出更適合的預測。

舉個簡單的句子為例：

I am starting to be quite busy next week and feel pressured finish it.

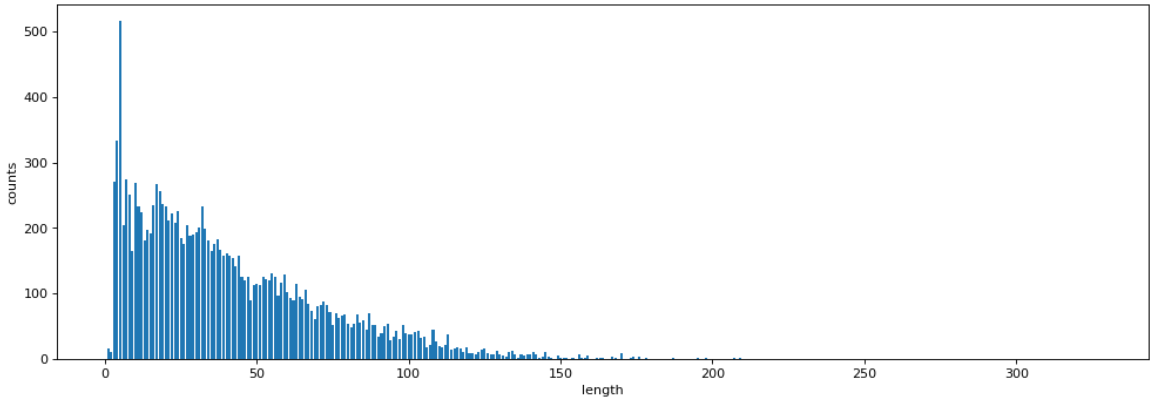
在沒有 attntion 機制的情況下，每個字的重要程度都相同，但在我的模型裡，會特別注重 pressured 這個字，還有些許的 feel，這能使得模型找到值得注意的地方，產生更相關的結果。

接著我檢視了模型的表現，我發現加入 attention 機制後，模型的確表現得比尚未加入 attention 機制前來的好，以下為我的模型中加入 attention 機制後，在 kaggle 上的分數：

Other imformation from dataset

1. 觀察句子長度

   下圖是我用 train_HW2dataset.csv + dev_HW2dataset.csv 所得到句子長度分布圖， 從圖中可以發現有許多句子的長度都是小於 50 個單詞，甚至在 0~10 個單詞的地方，有最多句子分布於此。



   我也在 encoding function 裡面測試了很多數值，這裡一樣是用 train_HW2dataset.csv 作為訓練資料，dev_HW2dataset.csv 作為驗證準確率的資料，以下為各數值所得到的準確率：

	accuracy
N = 10	48.6 %
N = 20	52.3 %
N = 30	42.3 %
N = 40	40.7 %
N = 50	39.1 %

對於為何我並未使用超過 50 的數值來做測試，是因為從上述的分布圖來看，可以發現有超過一半的句子的長度並沒有超過 50，再加上從 N = 30 開始，準確率就已經開始在下降，所以 N > 50 的情況，我便沒有再多加考慮。

2. 加入 speaker 考慮

   每個人可能會有自己獨特的說話方式、語氣，舉例來說，" I am proud of you." 光看字面的意思可能會認為是正面的、開心的(joy)句子，但有些人可能會用反諷的語氣來說出這句話，這時就會變成負面的、嘲諷的(disgust)句子，尤其在六人行這部影集裡，有許多句子都為嘲諷語氣，光看文字，可能造成誤判。

   我將所有 speaker 印出後發現除了主角的六個人之外，其餘的角色皆沒有說超過一百個句子，有些甚至只說過一句話，於是除了六個主角有自己的標記之外，我將其他角色一律標記成 7，此標記會在 encode 的時候一起加進 encoded 並回傳，以下為各角色各自被標記的號碼：

   ```
   def man_map(speaker):
       if speaker == "Ross":
           return 1
       elif speaker == "Joey":
           return 2
       elif speaker == "Rachel":
           return 3
       elif speaker == "Phoeba":
           return 4
       elif speaker == "Monica":
           return 5
       elif speaker == "Chandler":
           return 6
       else:
           return 7
   ```

   原本我認為加入 speaker 考慮會提升準確率，沒想到準確率不升反減，我認為可能是我 encode 的方式不對，speaker 可能無法和 text 放在一起訓練，可能要另外用其他方式來記錄每句話的 speaker 究竟為誰，或是需要透過其他方式來實作。

Comparison

以下表格是我將我的模型根據是否將句子進行 preprocessing、有無使用 pretrained word embedding、有無使用 attention 機制進行測試，其在 kaggle 上的分數如下：

		F1 score on kaggle
1	LSTM	0.22448
2	LSTM + preprocessing	0.19313
3	LSTM + pretrained word embedding	0.30153
4	LSTM + preprocessing + pretrained word embedding	0.29321
5	LSTM + attention	0.30483
6	LSTM + preprocessing + attention	0.27467
7	LSTM + pretrained word embedding + attention	0.32985
8	LSTM + preprocessing + pretrained word embedding + attention	0.31728

根據表格，比較 1 和 2、3 和 4、5 和 6、7 和 8，我們可以發現，先不考慮有無使用 pretrained word embedding 或是有無使用 attention 機制，只要將句子進行 preprocessing ，模型的表現就會降低。

接著，比較 1 和 3，這裡的 pretrained word embedding 是使用 glove.840B.300d.txt，可以發現使用 pretrained word embedding，模型的表現會好很多，F1 score 提升了不少。

接著，比較 1 和 5，使用 attention 機制會使模型的輸出找到值得注意的地方，產生更相關的結果，從 F1 dcore 我們可以發現，模型的表現確實好了很多。

最後，觀察 1、3、5、7，同時使用 pretrained word embedding 和 attention 機制，甚至比只使用其中一個的表現來的更好，這也是為什麼我最後將兩者都運用在我的模型中。

Difficulties and Solutions

1. 一開始在建構模型時，因為對於資料的維度並不是太熟悉，所以常常會遇上維度的錯誤，但寫了一下之後，就能漸漸掌握資料大致上會長什麼樣子，也就比較少遇上這些錯誤了。
2. 對資料進行 preprocessing 時，始終無法提升準確率，想了很久還是不知道該如何實作，但後來仔細想想，我發現那些 preprocessing 的方式，似乎真的會影響模型做判斷，這也難怪準確率一直無法提升，反倒還下降了些。
3. 在實作這次作業的過程中，中間有一段時間都只能用 cpu 來跑，一直不知道為什麼，所以每次跑一次都很花時間，結果後來發現是因為在下載 torchtext 的過程中，因為 pytorch 的版本不符，所以被自動升級了，但升級的版本為 cpu 的版本，後來降級回來，就能順利的用 gpu 執行了。